THE ROLL BACK CHIP: HARDWARE SUPPORT FOR DISTRIBUTED SIMULATION USING TIME WARP

UUCS-87-025

Computer Science Department
University of Utah
Salt Lake City, Utah
October, 1987

by
Richard M. Fujimoto, Jya-Jang Tsai and Ganesh Gopalakrishnan
Computer Science Department,
University of Utah,
Salt Lake City, UT 84112

† This work was supported by ONR Contract Number N00014-87-K-0184. and NSF Contract Number MIP-8710874

Contents

List of Figures

Abstract

Distributed simulation offers an attractive means of meeting the high computational demands of discrete event simulation programs. The Time Warp mechanism has been proposed to ensure correct sequencing of events in distributed simulation programs without blocking processes unnecessarily. However, the overhead of state saving and rollback in Time Warp is one obstacle that may severely degrade performance.

A special purpose hardware component, the rollback chip (RBC), is proposed to manage the state of a processor and provide an efficient rollback mechanism within a node of a parallel computer. The chip may be viewed as a special purpose memory management unit that lies on the data path between processor and memory. The algorithm implemented by the rollback chip is described, as well as extensions to the basic design. Implementation of the chip is briefly discussed. In addition to distributed simulation, the rollback chip may be used in other applications using the Time Warp mechanism, notably distributed database concurrency control.

1 INTRODUCTION

Discrete event simulation programs often possess computational requirements far exceeding the capabilities of the fastest available machines. For example, simulation of communication networks, digital logic networks, and large parallel processors often require hours or even days on conventional, uniprocessor machines[RF87]. One approach to solving this problem is distributed simulation — the execution of simulation programs on a parallel computer. A distributed, discrete event simulation program consists of a collection of autonomous *logical processes* that interact by exchanging timestamped messages. The seemingly high degree of parallelism that is present in many of the aforementioned applications combined with the recent emergence of multiple processor computer systems containing hundreds or thousands of high performance microprocessors has renewed interest in this approach.

However, a critical problem must be resolved by the distributed simulation program, namely, the management of simulated time. Each logical process must ensure that it only processes incoming messages in non-decreasing timestamp order. This is a difficult task because, in general, each process is uncertain as to what messages will be sent to it in the future. As a result, a process may be forced to wait until it can determine with absolute certainty the next message that should be processed. This situation, called *artificial blocking*, results from uncertainty of future incoming messages (in contrast to the usual notion of blocking in parallel programs that results from data dependencies in the computation), and can easily lead to deadlock.

The deadlock problem has been attacked by several researchers and distributed simulation algorithms based on deadlock avoidance or deadlock detection and recovery methods have been developed [CM79,DS80]. However, the artificial blocking problem remains, and empirical evidence indicates that these methods fail to achieve good speedup for many workloads that contain moderate or high degrees of parallelism [Fuj88,Ree87].

In contrast to these "conservative" distributed simulation strategies, the *Time Warp* approach uses an "eager" event (i.e., message) processing policy where received messages are processed as soon as the processor to which the logical process is mapped is available, independent of any messages that might arrive in the future [Jef85]. A roll back mechanism is used to recover from errors that might arise should events be processed in an incorrect timestamp sequence. An elegant mechanism called *anti-messages* is used to undo the effect of messages sent by the rolled back computation. Finally, one other important aspect of the Time Warp paradigm is the notion of global virtual time, or GVT. GVT provides a bound on the amount of computation that will have to be rolled back. This provides a mechanism to reclaim memory used to hold previously saved state information necessary for roll back.

The Time Warp mechanism may also be applied to other applications besides distributed simulation. Notably, Time Warp has been proposed for distributed database concurrency control and virtual circuit communication. Use of Time Warp in these applications are discussed in [Jef85].

The Time Warp approach offers great potential for good performance because it avoids *both* the artificial blocking and deadlock problems discussed above. However, Time Warp may still fail to achieve good speedup even for simulation programs exhibiting a high degree of parallelism because:

- Rollbacks may occur frequently — any rolled back computation represents time wasted by the processor.
- The overhead to allow rollback can be great — the state of the computation must occasionally be saved. This overhead must be incurred even if no rollback is necessary.

The former problem, frequency of rollback, is best tackled by appropriate scheduling techniques. This paper focuses on the second problem. We propose the use of *special purpose hardware* to minimize state saving and state management overhead. A memory management chip, called the *rollback chip* (or RBC) is proposed for this purpose. The RBC is a key component of a special purpose, distributed simulation engine based on the Time Warp paradigm that is currently being investigated.

The rest of this paper is organized as follows: The next section discusses related work. Section 3 is devoted to describing the RBC and the algorithm it uses for managing program state. Extensions to the proposed RBC design to increase its flexibility will then be discussed, followed by a discussion of issues related to incorporating the RBC in a design using off-the-shelf microprocessors that contain on-chip caches or memory management units. Finally, a prototype implementation that is currently under development is briefly described.

2 RELATED WORK

Use of parallel processing and special purpose hardware to improve the performance of simulation programs is not new. For example, numerous logic simulation engines have been constructed and are now available from commercial manufacturers that yield speedups ranging from a factor of 10 to 1000 (see [FWW84,SI86] for a taxonomy and survey of this work). However, work up to now has been restricted to continuous and *fixed* time increment paradigms. These are not appropriate for applications such as communication networks, high-level simulation of computer architectures, and queuing networks, to name a few. The focus of the current discussion lies in the use of special purpose hardware to be used in a high performance, discrete event simulation engine.

Work in speeding up *variable* time increment simulation has focused on the use of *general purpose* multicomputers and multiprocessors in distributed simulation. Numerous distributed simulation algorithms have been developed (e.g., see [JJE79,Mis86,Jef85]; a survey is presented in [Kau87]). Comfort has examined using dedicated processors to implement simulation primitives such as event list manipulation and random number generation [Com84,Com82,Com83,C*84]. However, the amount of parallel execution that can be obtained from such an approach is limited. Others have proposed running *independent* trials of the simulation program on separate processing elements [B*85]. While useful in Monte Carlo simulations where results from individual simulation trials can be combined to produce statistically significant results, this approach cannot be used in deterministic simulation, e.g., instruction-level simulation of parallel computer architectures driven by parallel application programs. Nevertheless, a special purpose simulation engine must be able to exploit such techniques whenever possible.

3 THE ROLL BACK CHIP

The envisioned distributed simulation engine is an extensible message-based parallel computer that features widespread use of ASICs (application specific integrated circuits) as well as more conventional components such as microprocessors and memories. No globally shared memory is used as it is not essential and would require an expensive interconnection switch. The logical topology of distributed simulation programs cannot be known a priori, so a symmetric hardware topology such as a mesh or hypercube should be used because they do not contain any inherent bottlenecks [RF87].

Each node of the envisioned system is a multiprocessor containing a general purpose microprocessor (GPP) with rollback hardware support (RBC), a communication processor, and various specialized processors (ASICs) as described below (see figure 1). The GPP executes application and system code and coordinates the operation of the ASICs. Shared memory is used *within* each node to facilitate tightly coupled interactions among the individual ASICs. An expensive interconnection switch is not required because the number of components within each node is limited. The communication processor provides hardware support for message-based communications. One

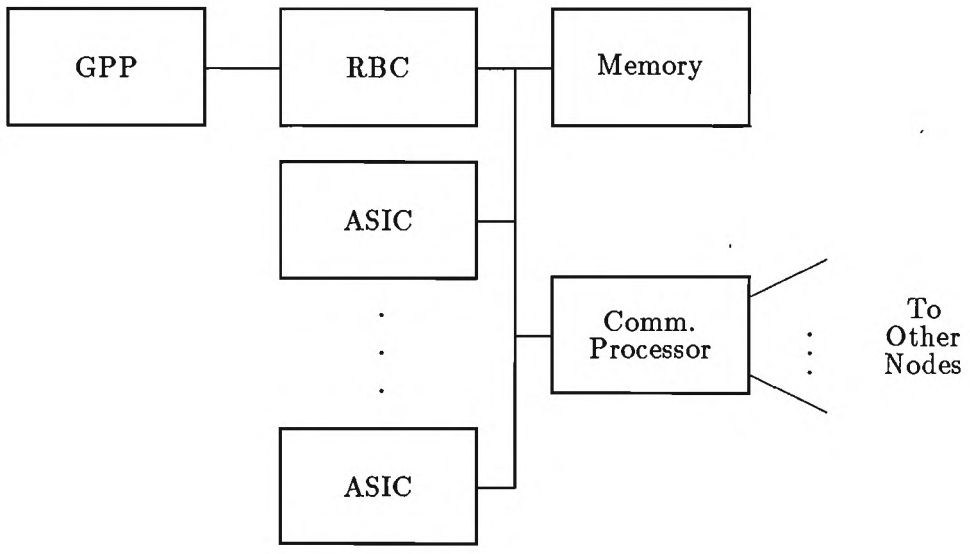

Figure 1: Configuration for Each Node of the System

design of such a component is described elsewhere[RF87].

The purpose of the roll back chip (RBC) is to provide an efficient mechanism to roll back computations within a single processor node under the control of the Time Warp program running on the node. It is the node's responsibility to determine when the state of the simulator running in it should be checkpointed, i.e., "marked", when rollback should occur, and how far the computation should be rolled back.

The roll back chip is, in effect, a memory management unit. It ensures that any state information which may have to be restored by a rollback operation is preserved. The state information is eventually discarded as global virtual time is advanced. Use of special purpose hardware allows otherwise time consuming operations such as roll back to be performed rapidly in a few clock cycles.

3.1 Functional Specification of the Roll Back Chip

A conceptual model of the storage management functions provided by the rollback chip can be described as follows: the rollback chip maintains different *versions* of state variables to enable a previous version to be restored when a rollback occurs. Different versions of the same variable are stored in separate storage areas called *mark frames*. The organization of the mark frames can be viewed, at least for the moment, as an unbounded stack. The *current mark frame* or *CMF* refers

to the frame at the top of the stack. The CMF register in the rollback chip contains a pointer to this frame.

It is the responsibility of the CPU to indicate when the current state of the processor should be preserved, and may require restoration as the result of a rollback. The CPU issues a *mark* operation to mark the current state as one which may have to be restored in a rollback. This operation pushes a new mark frame onto the top of the stack. The rollback chip ensures that subsequent memory write operations access the current mark frame so that older versions of the data remain preserved deeper in the stack. A rollback operation requires one to pop mark frames from the stack until the desired version of the simulator state is obtained. Each read operation requires a search through the stack to locate the most recent version of the data.

Of course, it is not realistic to assume that an unbounded stack can be used. According to the Time Warp paradigm, very old versions of the simulator can be discarded when their timestamp is smaller than *global virtual time* (*GVT*), that is, all except the most recent version preceding GVT. This process of reclaiming memory used by old state vectors is called *fossil collection*. Fossil collection is performed in the rollback chip by discarding mark frames from the *bottom* of the stack. To allow this storage to be re-used, the mark frames are organized as a circular list. In addition to the current mark frame register, the rollback chip also maintains a register called the *oldest mark frame* (*OMF*) register which points to the oldest mark frame that may have to be restored on a rollback operation.

3.1.1 Rollback Chip Operations The rollback chip supports six operations: reset, memory read, memory write, mark, rollback, and advance GVT. The CPU is responsible for generating these operations (via writes into control registers of the RBC chip) as governed by the Time Warp program which it is executing. The functional behavior of these operations are described below.

Reset Initialize the rollback chip prior to the execution of a Time Warp program.

Mark Mark the current state of the system as one which may have to be restored by a rollback operation.

Write (A,D) Write data D into memory address A. If the data at this location has been written since the last mark operation, then the most recent version of this data may be overwritten. Otherwise, the existing data at this location must be preserved.

Read (A) Read the most recent version of data associated with address A and return this data to the CPU.

Rollback (k) Restore the system state to that which existed just prior to the kth previous mark operation. This effectively undoes all modifications to state variables since the kth previous mark operation.

Advance GVT (k) The k oldest state vectors are no longer needed, and can be fossil collected.

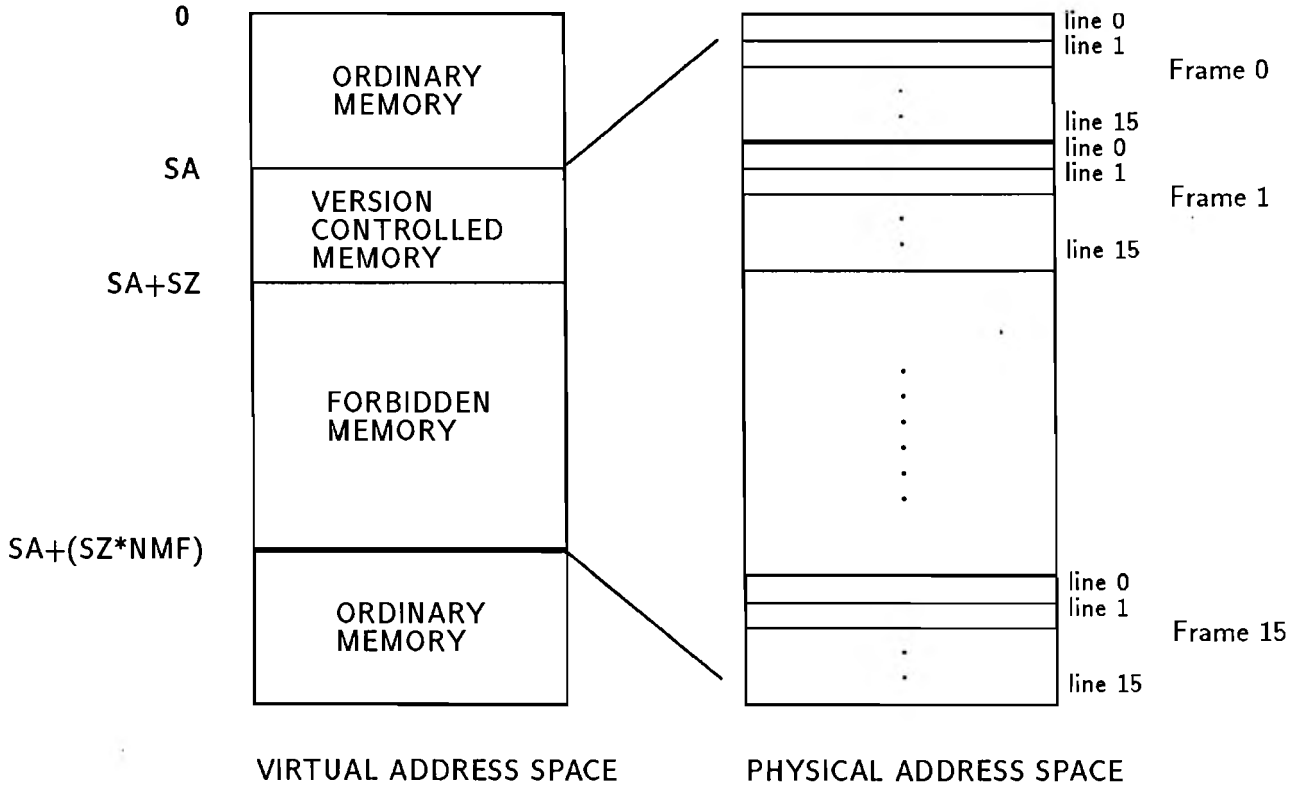

Figure 2: Virtual and Physical Address Spaces

3.1.2 The Memory Map The address space seen by the CPU contains four types of memory:

1. *Version Controlled Memory* contains state variables whose previous values can be restored via rollback operations.
2. *Ordinary Memory* is memory which is *not* version controlled, and therefore *cannot* be restored by the rollback chip to a previous state. Read and write accesses to ordinary memory have the same semantics as conventional random access memory. Code as well as any data areas which do not need to be restored on rollback are stored here.
3. *Forbidden Memory* is memory which is managed by the rollback chip and is not directly accessible to the CPU.
4. *Control and Status Registers of RBC.* These may be mapped anywhere in the ordinary memory address space.

The compiler must ensure that the appropriate type of memory is used by the simulation program, and that nothing is mapped to forbidden memory.

The memory map seen by the CPU is depicted in figure 2. We assume the memory is byte addressable. In figure 2 :

SA or starting address indicates the beginning address of version controlled memory.

SZ denotes the size in bytes, of a single mark frame. All mark frames are the same size, so frame zero begins at address SA, frame 1 at SA + SZ, etc.

NMF denotes the number of mark frames managed by the rollback chip.

Version controlled memory occupies addresses SA to SA + SZ - 1, and forbidden memory occupies address SA + SZ to SA + (NMF * SZ) - 1 inclusive. All other addresses refer to ordinary memory or RBC control or status registers. Read and write references to ordinary memory are passed by the rollback chip unmodified to the memory system.

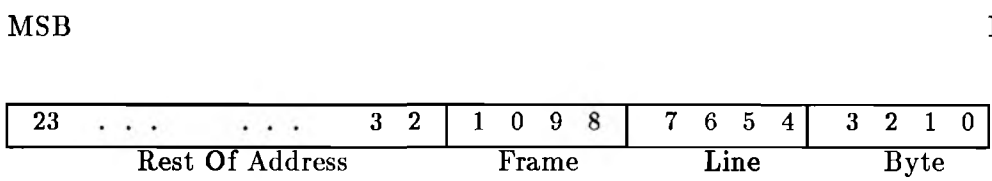

Figure 3: Address Format

3.1.3 Physical Addresses and Address Translation The memory map described above defines a *virtual address space* which is visible to the CPU. One can view the rollback chip as a special type of memory management unit which maps virtual addresses generated by the processor to physical addresses which are passed to the memory system. The *physical address space*, i.e., the memory map seen by the rollback chip, is essentially a more detailed version of the memory map described above.

The rollback chip subdivides the forbidden memory into mark frames, and each mark frame is again composed of a set of *lines* (see figure 2). As will become apparent momentarily, a line is similar to a line or block in a cache memory system.

A prototype rollback chip is currently under development. It supports 16 mark frames, each containing 16 lines, and each line contains 16 bytes. Each frame therefore contains 256 bytes, and the rollback chip manages a total of 4K bytes of memory. This chip is intended to demonstrate the concept. Rollback chips for practical applications would be expected to manage much larger amounts of memory. The discussion which follows will refer to the prototype chip in order to facilitate the presentation. Schemes for extending the current prototype will be discussed later.

For efficiency reasons, it is advantageous to assume that SA, the starting address of frame zero, is a multiple of NMF * SZ, i.e., a multiple of 4096 in the prototype chip. This ensures that

the 12 least significant bits of SA are all zeros. If one further assumes that the frame size is a power of two, then additions requiring carry propagation are *not* necessary to form physical memory addresses.

We will assume that memory operations do not cross line boundaries. Extending the rollback chip to allow memory accesses across line boundaries is straightforward, however, since such references can be viewed as two independent memory references to different lines.

The virtual address generated by the CPU is (say) a 24 bit memory address referring to ordinary or version controlled memory. If the reference is to ordinary memory, the physical address is identical to the virtual address, and is passed to the memory system unmodified. However, if the address refers to a location in version controlled memory, the rollback chip further subdivides the address into the following fields (see figure 3):

Byte the four least significant bits of the address. These indicate a byte within a line.

Line the next four significant bits of the address. These indicate a line within a mark frame.

Frame the next four significant bits of the address. The address generated by the CPU must have zeros in these four bits, or else an error is flagged by the RBC.

ROA or "Rest of Address", the remaining twelve bits of the address field. If these bits match the upper twelve bits of SA, then the address refers to version controlled memory. Otherwise, the memory request refers to ordinary memory.

The rollback chip translates the virtual address to a physical address by replacing the frame number field with either the current mark frame in the case of a write, or the frame holding the most recent version of the data in the case of a read. This address is then passed to the memory system and the memory operation is performed[1].

3.2 Informal Description of the Rollback Chip Algorithm

Before describing the implementation of the six rollback chip operations — reset, read, write, mark, rollback, and advance GVT — some preliminary discussion of an important data structure and an implementational problem must be described. These will be described next, followed by a description of each of the six rollback chip operations.

3.2.1 Written Bits Recall that the read operation must locate the mark frame containing the "most recent version" of the referenced line. The rollback chip must maintain some state informa-

[1] Actually the operation of reads and writes is slightly more complicated, but this address translation mechanism serves as a good conceptual model to understand the operation of the rollback chip.

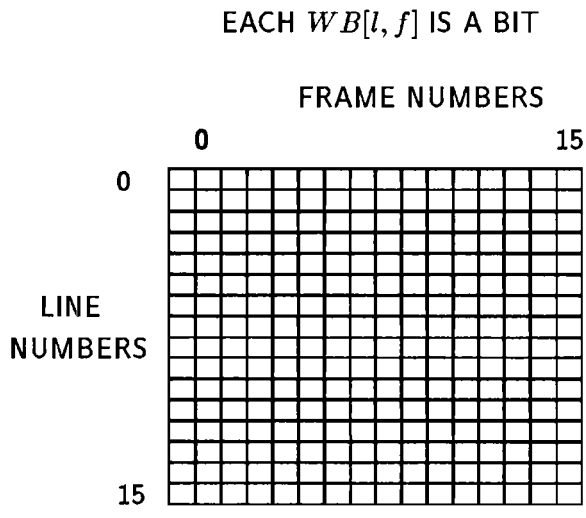

Figure 4: The Written-Bits Array

tion to identify this line. In particular, the rollback chip must keep track of which lines in each mark frame contain valid data, and which do not.

The rollback chip maintains an array of *written bits* (WB) for this purpose (figure 4). The written bits are stored within the RBC for quick access. A single written bit is associated with each line of each frame managed by the RBC. The written bits are organized as a two dimensional array: WB[l,f] is set whenever data is written into line l of mark frame f. Each column of bits in the array indicates the bits for a single mark frame, and each row the bits for a given line number across the various frames. A set written bit indicates that the data stored in that line in memory may be returned to the CPU. A cleared written bit indicates that no valid data is stored in the

```
FOR i:=0 TO NMF DO
        /* return frame number relative to CMF */
        IF WB[Line, (CMF-i+NMF) MOD NMF] = 1 THEN RETURN (i);
        END;
RETURN (ERROR);          /* all zero written bit is error state */
```

Figure 5: Program to Locate Most Recent Version of Line

line. The written bits are similar in function to the 'dirty bits' associated with pages in a paged virtual memory system.

When a memory read occurs, the most recent version of the data is found by searching the row of written bits corresponding to the referenced line, beginning with the current mark frame and proceeding in turn to older mark frames. The first set written bit encountered indicates the frame with the most recent version of the line. This operation is described by the program fragment shown in figure 5. A hardware error has occurred (or the chip has not been reset) if all of the written bits for a line are reset. The rollback chip is in an illegal state if this occurs.

The searching operation can be efficiently implemented in hardware as follows: first the row of written bits for the referenced line is read. The bits are then rotated to align them so that the CMF bit is in the rightmost bit position. Finally, the shifted bits are passed through a priority encoder with the rightmost bit assigned position zero and given highest priority. The resulting number indicates the frame relative to the current mark frame which holds the most recent version of the data. The most recent version of a line can thus be found using straightforward combinational logic.

3.2.2 The Seldom Written Data Problem The circular buffer implementation of the mark frames provides a very simple and elegant mechanism to implement fossil collection. However, it presents a problem which complicates the rollback algorithm somewhat. To gain some insight into this complication, consider an erroneous algorithm in which the mark operation increments the CMF pointer and clears all of the written bits corresponding to the newly acquired mark frame. From a conceptual standpoint, this is sensible because the purpose of the mark operation is to allocate a fresh frame on top of the stack. However, consider a piece of data which is written very seldom, e.g., only once during the entire computation. In this case, successive mark operations will eventually "wrap around" and clear the only written bit set for this line, erasing our only record of valid data for that line.

To circumvent this problem without abandoning the simple circular buffer mechanism, some data copying will be required. One approach is to impose the following constraint: all of the written bits in the frame pointed to by the OMF register must be set. This has the desirable property that it guarantees that valid data can always be found for the oldest frame to which we will ever have to roll back. However, this approach requires that a copy operation take place whenever OMF is advanced to a frame with one or more zero written bits. This will require seldom written data to be copied on virtually every advance GVT operation.

The rollback algorithm described below employs a lazy approach to copying. Data copying is deferred until the data which must be preserved for a possible future rollback is about to be overwritten. Data copying is also required in some circumstances to prevent the RBC from entering an invalid state. The lazy copying algorithm avoids unnecessary copying of seldom written data.

3.3 A Lazy Rollback Algorithm

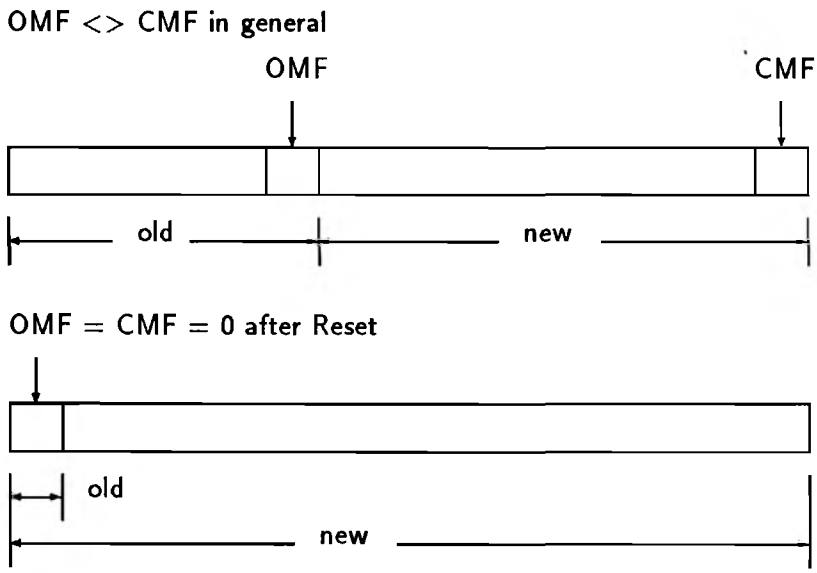

Figure 6: Definition of New and Old for a Line

state	old	new
00	=0	=1
01	=1	=0
10	=1	> 0
11	> 1	*any*

Figure 7: Line State

The principle feature of the lazy rollback operation is that it avoids excessive data copying of seldomly written data. In addition to the written bits, some state information is associated with each line to denote the state of that line. Two bits are required for each line, in contrast to the written bits in which one bit is required for each line *in each frame*. 32 state bits are required in the prototype chip since 16 lines are provided in each mark frame.

Consider the written bits for line l, i.e., row l of the written bit matrix. Let us define new_l as the number of written bits set in the frames more recent than the oldest mark frame, up to and

```
RESET:  WB[i,j] := if (j = 0) then 1 else 0;
        CMF := 0; OMF := 0;

MEMORY READ AT ADDRESS [roa | 0000 | line | byte]:
        Read M[roa | MRV | line | byte]

WRITE DATA D AT ADDRESS [roa | 0000 | line | byte]:
        Case 1: /* WB[line,CMF]=1 AND state[line] != 00 */
                Write D to M[roa | CMF | line | byte]
        Case 2: /* WB[line,CMF]=1 AND state[line] = 00 */
                Buffer := M[roa | CMF | line | 0000]
                Write Buffer to M[roa | OMF | line | 0000]
                Write D to M[roa | CMF | line | byte]
        Case 3: /* WB[line,CMF]=0 AND state[line] != 00 */
                Buffer := M[roa | MRV | line | 0000]
                Write D into Buffer
                Write Buffer to M[roa | CMF | line | 0000]
                WB[line,CMF] := 1
        Case 4: /* WB[line,CMF]=0 AND state[line] = 00 */
                Buffer := M[roa | MRV | line | 0000]
                Write Buffer to M[roa | OMF | line | 0000]
                WB[line,OMF] := 1
                Write D into Buffer
                Write Buffer to M[roa | CMF | line | 0000]
                WB[line,CMF] := 1
                WB[line,MRV] := 0

MARK OPERATION:
        IF ((CMF+1)=OMF) THEN
            ERROR: Ran out of Mark Frames
        FOR EACH LINE i:
            IF (state[i] = 10) AND (WB[line,CMF] = 1) THEN
                  buffer := M[roa | CMF | line | 0000]
                  Write buffer to M[roa | OMF | line | 0000]
                  WB[line,OMF] := 1;
                  WB[line,CMF] := 0;
                  CMF := (CMF+1) MOD NMF;
            IF (state[i] != 01)
                  WB[line,CMF] = 0;
                  CMF := (CMF+1) MOD NMF;

ROLLBACK k FRAMES:
        IF (OMF > (CMF - k)) THEN
                ERROR: Illegal Rollback
        ELSE CMF := CMF - k;

ADVANCE GVT k FRAMES:
        IF ((OMF + k) > CMF) THEN
                ERROR: Illegal Advance GVT Operation
        ELSE OMF := OMF + k;
```

Figure 8: Rollback chip operations

including the current mark frame (see figure 6). Similarly, define old_l as the number of written bits set in the remaining frames for line l, i.e., those frames older than OMF including the frame pointed to by the OMF register. Each line will always be in one of the four states listed in figure 7. The value of old_l must always be at least one in order to ensure that we can roll back to the OMF, the only exception being state 00 which is a special case designed to avoid unnecessary copying of seldomly written data.

The state information for a line is derived from the written bits for the line and the OMF and CMF registers. This information can be implemented with simple combinational logic embedded within the written bit array. This simplifies the design because the state information is updated automatically whenever the written bits, OMF, or CMF are modified, so no explicit action is required by the control unit in the rollback chip.

The six operations implemented by the rollback chip can now be described. These operations are described in terms of the state maintained by the rollback chip:

- The written bit $W[l, f]$.
- The state of line l denoted $state_l$. This is derived directly from the written bits for line l.
- The current mark frame register CMF.
- The oldest mark frame register OMF.

In the discussion which follows, MRV_l denotes the frame containing the most recent version of line l, and is obtained by scanning row l of the written bit array as discussed earlier. The memory address generated by the CPU for memory reads and writes contains the fields roa, frame (always 0), line, and byte as shown in figure 3. A register transfer level description of each of the six rollback chip operations is shown in figure 8.

3.3.1 The Reset Operation Before executing the Time Warp program, the rollback chip is initialized as shown in figure 8 via the reset operation. The written bits in frame zero are set even though no valid data has been written into the frame. This is to ensure each line always has at least one written bit set. Memory reads to uninitialized data thus yield arbitrary results just as they do in conventional computers. The state of each line becomes 01 after reset is complete.

3.3.2 The Read Operation The frame field of the memory address is replaced by MRV_{line} and the memory read operation is passed to the memory system. The state of the rollback chip is not changed.

3.3.3 The Write Operation As discussed earlier, the data must be written into the current frame. The actions taken by the RBC chip for a write operation fall into the following two broad categories:

1. If WB[line,CMF] is set and state $\neq$ 00 (case 1 in figure 8), we may simply write the data into the current frame. If WB[line,CMF] is not set and state $\neq$ 00 (case 3), the most recent version of the line must be read into a rollback chip buffer, modified, and then written into the current frame. This is necessary because the write need not modify the entire line. This situation is similar to a write miss in a cache memory system where the cache line must first be read before it is modified. Subsequent writes do not require this read step, so the first write into the line of a specific frame is always more expensive than subsequent writes.

2. If $state_{line}$ is 00 (old=0, new=1) then a copy operation is required because otherwise, the line would either change into an illegal state (old=0 and new >1) if the written bit in the current frame is not set, or data which may later have to be restored is overwritten if it is set. Therefore, the most recent version of the data should first be copied into the oldest frame, and the written bit for the oldest frame is set. The write into the current frame can then be performed. The state of the line becomes 10 (old=1, new>0).

The algorithm for a memory write is shown in figure 8. Buffer is a register in the rollback chip which holds a single line of data. After the first write is made to the line, subsequent writes up to the next mark operation use case 1, which would take only as much time as an ordinary memory write operation.

3.3.4 The Mark Operation The mark operation advances CMF by 1, modulo the number of mark frames NMF. This design of the rollback chip uses a lazy approach to fossil collection in that fossils are *not* collected when an advance GVT operation moves OMF past a frame. Instead, fossils are collected when the mark operation allocates a new frame. If a line in the new top of stack frame is no longer needed, the written bit is reset and the data stored is, in effect, fossil collected.

One special case may require a copy operation. Suppose the line is in state 10 (old=1, new>0) and the written bit of the newly allocate frame is set. The data cannot be fossil collected because it may be required in a subsequent rollback operation (note this isn't the case of old>1). The data must be copied to the oldest frame so it is available for a subsequent rollback.

Finally, the written bits for each line of the new frame are cleared, unless this is the only written bit set for that line. The written bit should therefore be cleared unless the line is in state 01 (old=1, new=0).

3.3.5 The Rollback Operation Recall the rollback operation contains a single parameter, the number of mark frames k to be popped off the stack. Rollback is implemented by simply decrementing CMF by k. A rollback which moves CMF beyond OMF is of course an error.

It is interesting to note that the written bits of the rolled back frames need *not* be reset! If old_l is at least one before the rollback, then the rolled back data becomes, in effect, very old data which will never be referenced again, and which will eventually be fossil collected by subsequent mark operations. If old_l is zero before the rollback, then the line must be in state 00 implying that new must be one. (It can be shown that the state old=0,new>1 is impossible.) In this case there is only one written bit set in the entire line, so we certainly do not want to clear it.

3.3.6 The Advance GVT Operation The advance GVT operation specifies a single parameter, the number of frames by which OMF can be advanced. This operation is implemented by simply incrementing OMF by k. An error occurs if OMF overtakes CMF. The written bits are not modified by this operation because the lines are not fossil collected until subsequent mark operations, in accordance with our lazy fossil collection approach.

4 EXTENSIBILITY

In the previous discussion, each RBC could only accommodate a limited amount of space for state variables, and a limited number of mark frames (i.e., versions of any individual variable). In practical applications, more and larger mark frames may be required. Extensions to the original RBC design to handle these situations will be described next.

4.1 Extentions for Larger Mark Frames

More state variables can be accommodated by replicating the RBC design. An n-fold increase in the amount of space for state variables can be obtained by using n RBCs in each node (see figure 9). Only simple address decoding circuitry is required to select the appropriate RBC when memory read and write requests are issued by the CPU. The reset, mark, rollback, and advance operations activate all RBCs simultaneously.

An alternative, more flexible, approach is to borrow cache memory ideas in the RBC implementation. Rather than implementing many *physical* RBCs as proposed above, one can envision several *virtual* RBCs mapped to, say, a *single* physical RBC. Each line managed by the physical RBC must be tagged to indicate the virtual RBC to which it corresponds. The address of each memory access must be compared with this tag to determine whether the desired virtual RBC is the one contained in the physical RBC. If so, a "hit" occurs and the memory request can be processed as usual. Otherwise, a "miss" occurs and the written bits for the desired virtual RBC must be loaded into the physical RBC.

For the time being, we assume only four virtual RBCs are used in the extended model. Define an array TAG[0..15], where each TAG[i] corresponds to *ith* line in WB. The value of TAG[i] could be either 0, 1, 2, or 3, indicating one of the four virtual RBCs. One more field must be added to

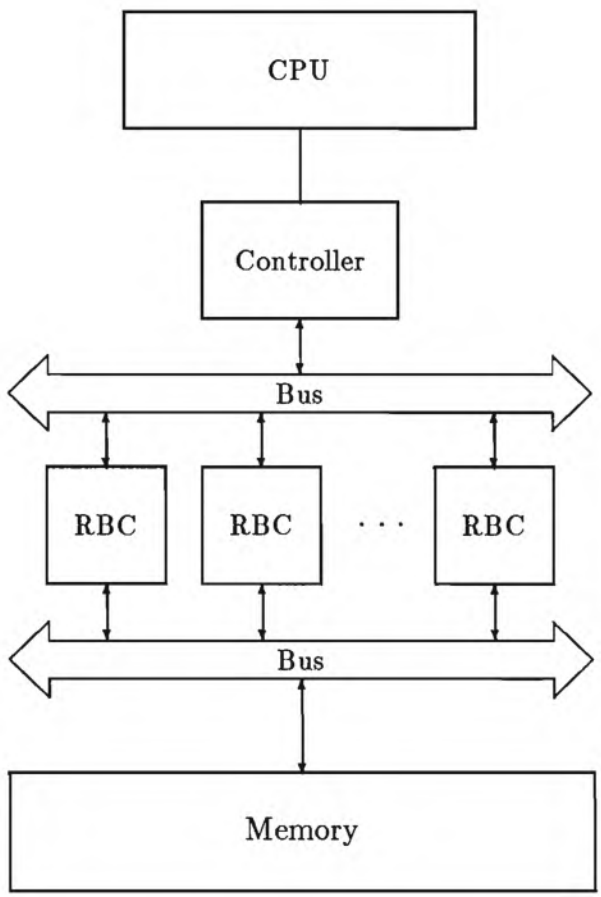

Figure 9: Multi-RBCs with a Single CPU

the address format shown in figure 3 to identify the virtual RBC to which the address refers. This new field occupies the two (assuming four virtual RBCs) least significant bits of the *roa* field. In general, use of n virtual RBCs produces n-fold increase in the address space of version controlled memory.

Ordinary memory locations are required for storing the written bit arrays of each virtual RBC. This memory is denoted by VC[0..3,0..15], each VC[i,j] contains 16 bits– the bit pattern of *jth* line in *ith* virtual RBC.

The modification to the original algorithm to accommodate virtual RBCs is shown in figure 10.

```
RESET:  Add
           For i=0 to 3
              For j=0 to 15
                 VC[i,j] := 1000000000000000; /* bit pattern */

MEMORY READ AT ADDRESS [roa | tag | 0000 | line | byte]:
        IF (TAG[line] == tag) THEN
             Read M[roa | tag | MRV | line | byte]
        ELSE
             VC[TAG[line],line] := WB[line];
             WB[line] := VC[tag,line];
             TAG[line] := tag;
             Read M[roa | tag | MRV | line | byte];

WRITE AT ADDRESS [roa | tag | 0000 | line | byte]:
        IF (TAG[line] != tag)
             VC[TAG[line],line] := WB[line];
             WB[line] := VC[tag,line];
             TAG[line] := tag;
        (the same as original algorithm)

MARK:
        For each line in WB
           VC[TAG[line], line] := WB[line];
        For i=0 to 3
           WB := VC[i];
           (the same as original algorithm);

ROLLBACK AND ADVANCE:
        (the same as original algorithm);
```

Figure 10: Extention for Larger Frames

The above scheme resembles a *direct address mapped* cache because each virtual RBC can be

mapped to only one physical RBC. *Set associated* RBC designs are also possible by using several physical RBCs and allowing a virtual RBC line to be mapped to any one. Address comparison hardware like that in set associative cache memories will be required, however.

4.2 Extensions for More Mark Frames

More mark frames are required if mark operations create new versions faster than advance GVT operations can fossil collect them. At some point, the number of required frames may exceed the number provided by the RBC. The RBC design can be extended to accommodate this situation.

Let us define the segment of memory managed by a *single* RBC to be a *working area.* The starting address of each working area is subject to the same alignment constraint as discussed earlier. As in the original RBC design, each working area is managed as a *circular list* containing 16 frames. To accommodate more frames, *multiple* working areas are allowed, and managed *in software* as an extensible list. The RBC manages only the most recently created working area(s). The RBC generates a trap to the processor when a reference to a working area not managed by the RBC (i.e., further back in the list of working areas) is required.

Initially, only one working area is in use. One new working area is allocated when the RBC runs out of frames, i.e., overflows the most recent (in this case, the original) working area. After the new area is allocated, the RBC manages the 16 most recent mark frames, *some of which are in the newly allocated working area, and the rest in the original.* As time progresses and more working areas are allocated, the RBC continues to span (at most) the two most recent working areas. These are referred to as the *current working areas* while older ones that are still in use are called *hidden working areas.* Written bits and other state information for hidden working areas are stored in arbitrary locations of non-RBC managed memory. One of the two current working areas is referred to as the even-numbered area, and the other as the odd-numbered area. The algorithm described in the original design can be viewed as a special case where all of the frames managed by the RBC belong to the same current working area, and no frames belong to hidden working areas.

This scheme for extending the RBC design allows the Time Warp program to expand to consume an arbitrary number of mark frames, subject only to the amount of memory available in the node. In order to manage the bookkeeping of the multiple working areas, the following registers are required:

WA[0..15] A 16-bit array, each bit corresponds to a frame in WB.

- WA[i] = 0, when the frame belongs to an even-numbered working area;
- WA[i] = 1, when the frame belongs to an odd-numbered working area;

ODD A register to indicate the current odd-numbered working area.

EVEN A register to indicate the current even-numbered working area.

OVER A register to indicate the point where the overflow occurs, OVER is also the oldest frame of each working area (the same for all mark frames).

NUM A register to count the total number of frames currently in use.

WB_AUX[i, j] An array in memory to hold the written bits for hidden working areas. This array is manipulated by processor routines. The indices i and j denote the j-th frame in the i-th working area.

WB in the RBC contains the written bits of portions of the current odd and even-numbered working areas. The meaning of OMF is also changed. It now points to the oldest mark frame, in the current working areas. The MRV (most recent version) is now determined not only by the frame number, but also by the working area (odd or even) containing the frame. The algorithm is such that the most recent version of each state variable is always within the current working areas.

The operations of the RBC are modified as follows:

Mark operation If the CMF does not exceed the OMF, i.e., no overflow of the current working area occurs, the mark operation is identical to that described earlier. Otherwise, the OMF is deleted from the RBC, i.e., its written bits are saved in WB_AUX. Usually, this will create a vacant mark frame that held state variables for the older of the two current working areas, so the tag bit associated with the frame (WA) must be complemented to indicate the frame now belongs to the newer area. If, however, the vacancy held variables from the newer frame (this occurs when all of the frames managed by the RBC are within the *same* working area) then a new working area must be allocated and the EVEN/ODD registers must be updated appropriately. The MARK operation can then be completed according to the algorithm described in the original RBC design, with one additional step: if updating the written bits would cause a line to have *no* written bit set in any frame in the current area, a copy operation is necessary to prevent the line from entering an illegal state. This allows all future read and write requests to be handled completely within the RBC without referring to "hidden" written bits stored in WB_AUX.

Rollback operation If the rollback does not go beyond the extent of WB, this operation is identical to that described earlier. If it does extend beyond this point, the RBC determines which working area is to become the most recent, restores the bit patterns of the working area into WB, and updates the values in EVEN, ODD, CMF and OMF to reflect the state of the computation after roll back.

Advance operation When the advance operation does not extend beyond the frames in the hidden working areas, the RBC simply decreases the counter NUM. Old working areas can now be fossil collected, and their storage reused. If the advance operation extends into the current working areas, the OMF register must be updated.

The details of the extended algorithm are given in figure 11. With the exception of the one special case noted in the MARK operation, program data is not moved in memory. The only swapping that must be done is the written bit information. This greatly reduces saving and restoring overhead required in the extended algorithm.

The address format need not be changed to accommodate the change of algorithm. As before, the RBC used the roa field of the address supplied by the CPU to determine if the reference is to version controlled memory. However, rather than passing this field unmodified to the memory system, it *replaces* it with the high order bits of the pointer to the working area (odd or even) that is referenced. These addresses are buffered in the RBC and updated when overflow and rollback occur.

5 CACHE AND MEMORY MANAGEMENT UNITS

This section will describe the interactions between RBC, cache memory and memory management units, features common to many off-the-shelf microprocessors. Ideally, the RBC should be integrated into a custom designed CPU, tailored to the operations performed by the RBC. However, it can be used with standard microprocessors if certain precautions are taken.

Caches are high speed memory devices that buffer frequently used data. They are common among high performance mainframe and mini computers, and more recently, among high performance microprocessors. When used with a microprocessor with an on-chip cache, the cache memory must necessarily reside between the CPU and RBC.

Caches use either a *write through* or a *write back* policy. If a write through policy is used, memory writes are transmitted through the cache and to the RBC as they occur. If a write back cache is used, the cache does not generate memory writes until cache lines are removed from the cache by the replacement policy. In the latter (write back) case, the RBC may not detect the write until long after it has occurred. This is problematic because a MARK operation may have occurred after the CPU generated the write. Therefore, the RBC may receive the write and MARK operations in the wrong order, causing an error.

There are several solutions to this problem. The most straightforward is to disable the cache or ensure that state variables of the simulation are not cached. Alternatively, one may flush the cache before each mark operation. The approach to be taken is dependent on the operation of the cache in the microprocessor being used. Caches must provide some facility to handle such situations in order to ensure consistency when I/O devices access memory.

The aforementioned problem does not arise if a write-through cache policy is used. However, regardless of the write policy in use, a rollback operation must flush the cache. This is necessary to prevent rolled back data from being read by the processor.

Similarly, many modern microprocessors contain a memory management unit (MMU). The

```
RESET:
       Add
            WA[i] := 0, for all i;
            EVEN := 0;
            ODD := -1;
            NUM := 1;

READ & WRITE:
        Unchanged (except the MRV is now determined by scanning the WB,
        WA and the value in EVEN/ODD)

MARK:
        IF (((CMF+1) mod NMF) == OMF) THEN
            IF (EVEN <= ODD) THEN
                WB_AUX[EVEN,OMF] := WB[OMF];
            ELSE
                WB_AUX[ODD,OMF] := WB[OMF];
            NUM := NUM + 1;
            For j = 0 to 15 do
                IF ( WB[OMF, j] ==1)
                   IF (WB[OMF+1, j] ==0) THEN
                        Buffer := M[roa | OMF | line | 0000]
                        Write buffer to M[roa | OMF +1 | line | 0000]
                        WB[OMF+1,j] := 1;
                   WB[OMF, j] := 0;
            end-do;
            IF(WA[0..15] == 0)  ODD := ODD + 2;
            IF(WA[0..15] == 1) EVEN := EVEN + 2;
            OVER := OMF;
            WA[OMF] := 1 - WA[OMF];
            OMF := (OMF+1) mod NMF;
            CMF := (CMF+1) mod NMF;
         ELSE
            (the same as previous algorithm, NUM := NUM + 1)

ROLLBACK k FRAMES:
        IF ( k > NUM ) THEN   ERROR: Illegal rollback;
        ELSE  IF (k > ft) THEN  /* ft = ((CMF-OVER+NMF) mod NMF)+1, the
                      number of frames in top working area */
                IF ((fs==0) and (OVER !=OMF)) THEN /* fs=((k-ft) div NMF)*/
                   j := OVER;
                   IF (EVEN > ODD) THEN
                      While ((j mod NMF) != OMF) do
                        WB[j] := WB_AUX[ODD, j];
                        WA[j] := 1;
                        j := j+1;
                      end-do;
                      EVEN := EVEN -2;
```

```
            ELSE
               While ((j mod NMF) != OMF) do
                  WB[j] := WB_AUX[EVEN, j];
                  WA[j] := 0;
                  j := j+1;
               end-do;
               ODD := ODD -2;
         ELSE
            For j = 1 to (fs+1) do
               IF (EVEN > ODD) THEN
                  EVEN := EVEN - 2;
               ELSE
                  ODD := ODD - 2;
            end-do;
            IF (EVEN > ODD) THEN
                  WB := WB_AUX[EVEN];
                  WA[0..15] := 0;
            ELSE
                  WB := WB_AUX[ODD];
                  WA[0..15] := 1;
          CMF := (OVER - (k -ft-fs*NMF) +NMF) mod NMF;
          NUM := NUM - k;
          IF((CMF -OVER+NMF) mod NMF +1 <= NUM) THEN
              OMF := OVER;
          ELSE
              OMF := (CMF-NUM+1+NMF) mod NMF;
       ELSE
            CMF := (CMF-k+NMF) mod NMF;
            NUM := NUM - k;

ADVANCE GVT k FRAMES:
        IF (NUM < k) THEN ERROR: Illegal advance;
        ELSE  IF ((NUM-fw) >= k) THEN /* fw=((CMF-OMF+NMF) mod NMF)+1 */
                  NUM := NUM - k;
              ELSE
                  OMF := (OMF+(k-(NUM - fw))+NMF) mod NMF
                  NUM := NUM - k;
```

Figure 11: Extention for More Frames

purpose of the MMU is to translate virtual addresses into physical addresses. If the RBC can be placed between the processor and the MMU, no difficulties arise. Again, this is not possible if the MMU is on the same chip as the CPU. If the RBC lies between the MMU and memory, the RBC deals with physical (rather than virtual) memory addresses. The RBC assumes, however, that the state variables occupy contiguous memory locations, so some constraints must be placed on the MMU mapping to ensure that this condition is not violated.

6 IMPLEMENTATION OF THE RBC

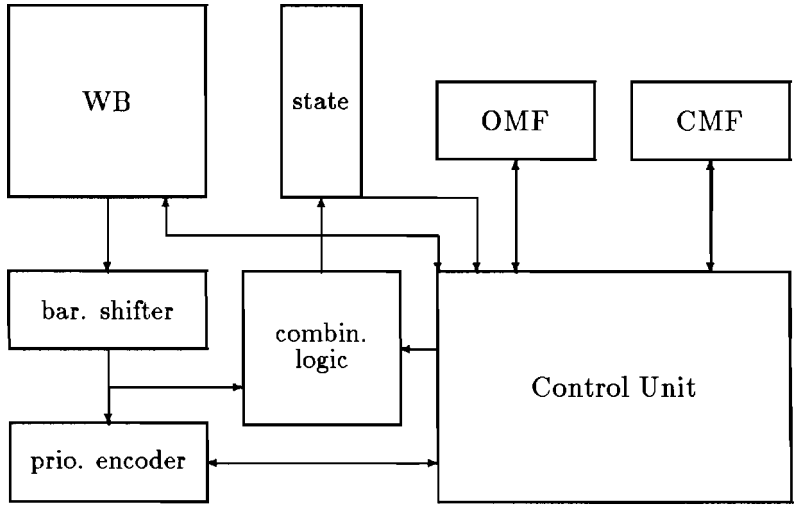

Figure 12: Functional Block Diagram

A block diagram of the original envisioned RBC design is shown in figure 12. The barrel shifter and priority encoder are used to locate the most recent version of a state variable. The value produced by the priority encoder is subtracted from the CMF register to obtain the appropriate mark frame. This operation can be implemented with combinational logic, allowing rapid computation of the most recent version's frame.

The combinational logic below the state bits is a circuit using input from the barrel shifter to determine the state of each row in WB. Alternatively, this can be implemented within the written bits array itself. Finally, the control unit plays a central role in controlling the operation of the chip.

7 CONCLUSION

We have described a special purpose component, the rollback chip, to offload state saving and version management overhead in the Time Warp algorithm to special purpose hardware. It is a key component of a special purpose, distributed discrete event simulation engine based on the Time Warp paradigm. Other aspects of the simulation engine are currently under investigation.

At the time of this writing, the algorithm used by the RBC has been formally verified. Detailed design and implementation of the RBC is currently in progress. Fabrication of key components of the RBC is expected to take place in 1988. Performance evaluations of the RBC, and the distributed simulation engine as a whole, are planned.

References

[B*85] W. Biles et al. Statistical Considerations in simulation on a Network of Microcomputers. *1985 Winter Simulation Conference Proceedings*, 388–393, December 1985.

[C*84] J. C. Comfort et al. The Design of a Multi-Microprocessor Based Simulation Computer - III. *Proceedings of the Seventeenth Annual Sumulation Symposium*, 227–241, 1984.

[CM79] K.M. Chandy and J. Misra. Distributed Simulation. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[Com82] J. C. Comfort. The Design of a Multi-Microprocessor Based Simulation Computer - I. *Proceedings of the Fifteenth Annual Sumulation Symposium*, 45–53, 1982.

[Com83] J. C. Comfort. The Design of a Multi-Microprocessor Based Simulation Computer - II. *Proceedings of the Sixteenth Annual Sumulation Symposium*, 197–209, 1983.

[Com84] J. C. Comfort. The Simulation of a Master-Slave Event Set Processor. *Simulation*, 42(3):117–124, March 1984.

[DS80] E. W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[Fuj88] R. M. Fujimoto. Performance Measurements of Distributed Simulation Programs. In *1988 Society for Computer Simulation Multiconference, San Diego, CA*, February 1988.

[FWW84] M. A. Franklin, D. F. Wann, and K. F. Wong. Parallel Machines and Algorithms for Discrete-event Simulation. *Proceedings of the 1984 International Conference on Parallel Processing*, 449–458, August 1984.

[Jef85] D.R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[JJE79] J.K.Peacock, J.W.Wong, and E.G.Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, February 1979.

[Kau87] F. J. Kaudel. A Literature Survey on Distributed Discrete Event Simulation. *Simuletter*, 18(2):11–21, June 1987.

[Mis86] J. Misra. Distributed Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Ree87] D. A. Reed. Parallel Discrete Event Simulation: A Shared Memory Approach. *To appear, IEEE Transaction on Software Engineering*, 1987.

[RF87] D. A. Reed and R. M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing. Computer Science*, MIT Press, 1987.

[SI86] R. J. Smith II. Fundamentals of Parallel Logic Simulation. *23rd Design Automation Conference*, 2–12, June 1986.